ASSIGNMENT OF BACHELOR'S THESIS

Title:	Cryptanalysis of RSA based on factorization
Student:	Petr Horák
Supervisor:	Mgr. Martin Jureček
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2020/21

Instructions

The security of many public-key cryptosystems relies on intractability of the integer factorization problem. The RSA encryption scheme belongs to the most common in this class of cryptosystems. Most attacks on RSA cryptosystems are based on factorization of integers or on exploitation of inappropriate parameters setting (e.g., low private exponent, common modulus, ...).

Survey some of the factorization techniques (especially Pollard's rho and (p-1) methods, Fermat's factorization, and the Quadratic sieve) and analyze their expected computational complexities.
Implement these methods or use existing implementations (e.g., Magma) and perform factorization-based attacks using various sizes of RSA public moduli.
Evaluate the performance results and discuss practical runtime complexities of these attacks.
Describe, implement, and evaluate at least three attacks on the RSA based on inappropriate parameters settings.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Head of Department	Dean

Prague November 29, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Cryptanalysis of RSA based on factorization

Petr Horák

Department of Computer Security and Information technology
Supervisor: Mgr. Martin Jureček

June 1, 2020

Acknowledgements

I would like to thank my supervisor, Mgr. Martin Jureček, for his guidance and all the help he has provided not only during the creation of this thesis.

I would like to thank Ing. Eliška Šestáková for her useful comments. I would also like to thank my friends, Stanislav Koleník, Radek Šmíd and Petr Hoffmann, for their support and encouragements.

Declaration

Citation of this thesis

Horák, Petr. *Cryptanalysis of RSA based on factorization.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Tato bakalářská práce se zabývá kryptoanalýzou RSA založenou na faktorizaci celých čísel. Rozebrané jsou vybrané faktorizační metody – důraz je kladen zejména na popis jejich časové náročnosti a porovnání z hlediska použitelnosti v reálných podmínkách. Kromě toho se práce věnuje i problémům s bezpečností RSA, které mohou vzniknout při neopatrném využívání této kryptografické metody. Vybrané metody jsou implementované za použití Magmy pro testování reálných výpočetních rychlostí na vzorových datech.

Klíčová slova RSA, RSA problém, faktorizace, Pollardova rho metoda, Pollardova p-1 metoda, Fermatova faktorizace, Dixonův algoritmus, kvadratické síto, Magma, společný modul, nízký soukromý exponent

Abstract

This bachelor's thesis deals with RSA cryptanalysis based on integer factorization. Described are some of the factorization methods. Emphasis is laid on their computational complexities and comparison in real-life scenarios of use. Another topic in this work is a discussion of security issues of RSA which can occur by incautious use of this cryptographic method. Selected algorithms are implemented in Magma language for speed testing on the sample data.

Keywords RSA, RSA problem, factorization, Pollard's rho method, Pollard's p-1 method, Fermat's factorization, Dixon's algorithm, quadratic sieve, Magma, common modulus, low private exponent

Contents

List of Figures

List of Tables

Introduction

Digital communication is a widely spread phenomenon we use on a daily base. Processing information via a cable or wirelessly became an essential thing in our lives. However, along with it came the problem of securing these communications and protecting against possible information thefts. Thus, many encryption methods came to use. Some of them are based on the integer factorization problem – RSA belongs to the most popular among them. However, all encryption methods have some issues when they are not used correctly.

There are several problems in RSA encryption schema. One of these is the choice of the modulus. Using various factorization algorithms, some numbers are vulnerable against a brute force attack. The most important parameter of these methods is computational complexity. This thesis gives much attention to this part, describes selected algorithms by the perspective of feasibility regarding their complexity demands. Studying these methods helps us avoid potential problems.

Some of the attacks on the RSA scheme are based on the exploitation of inappropriate parameters. Some of them are just less suitable than others. Description of problematic parameters and the threats for the cryptosystem are also parts of this thesis.

The primary aim of this thesis is to introduce specific factorization methods from the perspective of mathematical principles they are based on and their computational complexities. Another aim is to produce a study material for the course Mathematics for Cryptology (MIE-MKY)[1] and its alternatives for Czech students and remote study. The thesis also contains time tests of the selected methods using Magma. Magma is a software package created at the University of Sydney for mathematical computations with its own programming language.

The thesis is divided into four logical parts. The first deals with terminology and some definitions that will be used later on. The second part is about

[1]course taught at FIT CTU in Prague

specific factorization algorithms and their description and analysis. Follows a section about attacks on RSA which do not use factorization. The last part describes results of testing of factorization methods and selected attacks on RSA.

Chapter 1

State-of-the-art

Cryptography has been developing for a very long time. From antique and *Caesar cipher* to the modern age of advanced cryptosystems. This chapter contains a brief retrospective view on factorization. The end is dedicated to comparison with existing literature with a similar focus.

As far as the history of encrypting goes, there always exist efforts to decrypt secret messages. From the age of computing in hand, we got to the era of computers. Thus, the requirements for the cryptosystems to be unbreakable are not easy to fulfil. There is one more request, encryption and decryption with known parameters have to be performed quickly.

Modern cryptography is based on solid mathematical arguments. They usually use some kind of irreversible or slowly reversible math operations, such as multiplication and factorization as its inverse as was used in the cryptosystem RSA invented by Rivest, Shamir and Adleman in 1977. But the actual factorization problem is much older and so are the attempts to solve it.

First factorization algorithm was published and described by Leonardo of Pisa, better known as Fibonacci, in 1202 in his first book *Liber Abaci* [1, p. 20]. From then on, many great mathematicians took their part in building the history of factorization. In 1674, Pierre de Fermat introduced his algorithm to factorize numbers [1, p. 24] which is of good use in modern methods, especially with its enhancement from Maurice Kraitchik in the 1920s [2, p. 1474]. Works of Leonhard Euler and Adrien-Marie Legendre are also important.

Some new methods were published in the second half of the 20th century in works of J. M. Pollard, who introduced two of them [3] [4]. John D. Dixon continued with Fermat's work and in 1981, he presented a new algorithm [5]. In the same year, Carl Pomerance came up with another improvement and created *quadratic sieve* [6], one of the widely used algorithms. The idea of factoring using elliptic curves, brought by Hendrik Lenstra in 1985 [7], reaches very similar asymptotic running time to *quadratic sieve* but is especially fast when the number to be factored has some small factors. The last significant improvement in the field of factoring was made in 1988, after John Pollard suggested using algebraic number fields [2, p. 1479] and thus *number field sieve*, the fastest algorithm for factorization so far, was developed.

Even with modern computer networks, tremendous computing speed and optimized parallel algorithms, the biggest RSA key that was factorized until now is *RSA-250* with 250 decimal digits (829 bits). They used open-source *CADO-NFS* software, which is C/C++ implementation of *number field sieve*, and it finished in February 2020 [8] [9]. For the comparison, NIST[2] recommends the length of the public modulus of RSA to be 2048 bits or more [10, p. 15].

The main content of this thesis is a description of some factorization algorithms. There are many books and articles with a similar purpose. It is not possible to list all of them here, but some should be mentioned.

A great introduction to the world of cryptosystems is *Handbook of Applied Cryptography* [11]. There are descriptions of the same methods as in this thesis, and for the purpose of basic introduction, they work completely fine. However, in some points, they unnecessarily differ from the original papers. Also, there are some facts stated that are not very intuitive and can come as a surprise to a reader. The same points go for [12] too.

On the other hand, there is publication *Prime Numbers and Computer Methods for Factorization* [13]. It contains an enormous amount of factoring algorithms with detailed descriptions. However, for the purpose of introduction of the basic concepts, the book is far too advanced and full of not so important information.

The difference of this thesis is that it tries to achieve comprehension of these algorithms in a logical way along with trying to minimize diversion from the original papers. The purpose is not to make all the math arguments laying under the theory of these methods, as some of them are far beyond the scope of this thesis, but to introduce them in the logical and consistent way along with some basic mathematical background.

[2]National Institute of Standards and Technology

Chapter 2

Concepts

This thesis uses some non-trivial mathematical principles, definitions and notations. Here, the most important of them for the purposes of the later chapters are described.

2.1 Notation

Notation in this thesis follows original articles of the authors of the presented algorithms. If it not suitable, notation from [11] is used. However, if some redundant designation is used in order to make algorithms clearer, the notation is chosen accordingly for maximal consistency.

Describing algorithms in terms of computational complexity requires some kind of notation. For the purpose of this thesis, *big $\mathcal{O}$ notation* is used for the upper bound of the computational complexity functions, which are obviously non-negative.

Definition 1 (Big $\mathcal{O}$ notation)**.** *Function $f(n)$ is of the same or lower order than a function $g(n)$, written as $f(n) = \mathcal{O}(g(n))$, if function $f(n)$ is lower (or equal) than a constant multiple of $g(n)$ except for the finite amount of values.*

$$f(n) = \mathcal{O}(g(n)) \iff (\exists c \in \mathbb{R}^+)(\exists n_0 \in \mathbb{N}^+)(\forall n > n_0)(f(n) \leq c \cdot g(n))$$

In this thesis, *big $\mathcal{O}$ notation* is used to describe the amount of polynomial time operations, such as division or multiplication, depending on the size of the parameter n. As it is often very difficult to find the average computational complexity, only the worst-case scenario is taken into the account unless told otherwise.

There is one more complexity notation used in this thesis, *L-notation.* It is very useful for a description of some algorithms. For the purpose of this thesis, a reduced version of this notation, taken from [6, p. 90] is used.

$$L(n) = e^{\sqrt{\ln n \ln \ln n}}$$

2.2 Terminology

By basic rule of arithmetic, every integer is decomposable to the multiplication of primes, which are called its prime factors. To say that each of them is lower than a specific bound B, let's define smoothness in the following way.

Definition 2 (Smoothness)**.** *Let a be a natural number. We say a is B-smooth (or just smooth if it is obvious from context or discussed generally) if every prime factor p_i of a is equal or lower than some natural number B.*

2.3 Modular arithmetic

Almost the entire thesis is about modular arithmetic. There are two equivalent notations used.

$$a = |b|_n \iff a \equiv b \pmod{n}$$

Let's state (without proofs, they can be found in cited sources) some of the most significant theorems. Formulations taught on FIT CTU in Prague are used, specifically [14, slides 5 and 12] and [15, slide 12].

Theorem 1 (Little Fermat's Theorem)**.** *Let p be a prime and $a \in \mathbb{N}^+$ such natural number that is coprime with p. Then it holds*

$$a^{(p-1)} \equiv 1 \ (mod\ p).$$

As has been proven, it works, with some modifications, for all natural numbers. The following theorem is its generalization.

Theorem 2 (Euler's Theorem)**.** *Let $a, p \in \mathbb{N}$ be some coprime numbers. Then*

$$a^{\varphi(p)} \equiv 1 \ (mod\ p).$$

Another topic used in the thesis is solving system of congruences. Thus, following theorem come to use.

Theorem 3 (Chinese Remainder Theorem)**.** *Let's have the system of linear congruences*

$$\begin{gathered} x \equiv a_1 \ (mod\ p_1) \\ x \equiv a_2 \ (mod\ p_2) \\ \vdots \\ x \equiv a_N \ (mod\ p_N), \end{gathered}$$

where the modules p_i are pairwise coprime. Then the solution of this system exists and all solutions are congruent modulo P, where

$$P = \prod_{i=1}^{N} p_i.$$

2.4 Primes

Distribution of prime numbers among integers is one of the most mysterious problems in the world of mathematics. As it is connected to *Riemann Zeta function*, it is even included in the *Millennium Problems* list as the *Riemann Hypothesis* [16]. However, the asymptotic amount of primes lower than a certain bound is known.

Theorem 4 (Prime Number Theorem)**.** *Let $\pi(n)$ be the number of primes lower than some integer n. Then it holds*

$$\lim_{n\to\infty} \frac{\pi(n)}{\frac{n}{\ln n}} = 1.$$

Despite the fact, that $\pi(n) \sim \frac{n}{\ln n}$ is not the best-known approximation [17], it is sufficient for the purposes of this thesis.

Chapter 3

RSA

As usage and mechanisms of cryptosystem are crucial for breaking it, this chapter contains a basic description of the RSA – from the introduction and the concept of public-key cryptography to proof of the correctness of communication mechanism and security aspects.

3.1 Introduction

The most used public-key cryptosystem is the RSA [11, p. 285]. It was created in 1977 by Ronald L. Rivest, Adi Shamir and Leonard Adleman. In general, public-key cryptosystems (sometimes also referred to as asymmetric cryptosystems) use a key pair, two separate keys – public and private.

The public key may be widely available. It is used for data encryption. The only problem here is to make sure it really belongs to the presented subject. One way to approach it is to use certification authorities. The private key is held in secret, and it is used for decryption and signing messages.

The idea here is that knowing the public key will not help to find the private one in a reasonable amount of time. That is achieved via constructing public key using a one-way function, like multiplication (factorization) and modular exponentiation (discrete logarithm). Another thing to care about is a bijection of this function. Otherwise, there could be some issues concerning security.

When transmitting data, we can use the private key to sign it. The information held in the message can be compromised because everybody with access to the public key can read it. But the main feature here is the authenticity of the message, to alter it, one would have to know the private key of the transmitter.

If we want to ensure that information included in the message are secured, we can encrypt transmitted data with the public key of the receiver. Therefore, the only way to read them is to decrypt it with the private key.

3.2 Communication mechanism

To create private and public keys, the following steps must be undertaken:

1. Choose two prime numbers p and q with the appropriate length. The methodology for selecting these numbers is discussed in [18] and [19].
2. Compute public modulus: $n = p \cdot q$
3. Compute Euler's function of the public modulus: $\varphi(n) = (p-1) \cdot (q-1)$
4. Choose private exponent d: $1 < d < \varphi(n)$ and $GCD\,(d, \varphi(n)) = 1$
5. Compute e which is multiplication inverse of d in modulus of $\varphi(n)$: $ed \equiv 1 \pmod{\varphi(n)}$

Private key contains n, d. The public key is composed of n and e. This is the original procedure as was described in [20, pp. 122–123]. However, nowadays a different selection method is recommended, it is described in [18, pp. 50–53] and [19, pp. 33–35].

Let us suppose Alice wants to write a secret message to Bob. The first thing to do is to divide the content of the message to blocks and convert it to integers m_i, each of these must fulfil the condition $m_i < n$. Otherwise, m_i could be reduced by modulus, and that would lead to duplicates making decryption rather confusing. A ciphertext c_i is then computed in the following way:

$$c_i = |m_i^e|_n$$

Theorem 5 (Correctness of RSA)**.** *Let c_i be a ciphertext incurred from encrypting plaintext m_i by RSA using a public modulus n and an exponent e. Assume that d is inverse of e in modulus of $\varphi(n)$. Then:*

$$m_i = \left|c_i^d\right|_n$$

Proof. Let's rewrite c_i according to the definition stated earlier.

$$\left|c_i^d\right| = \left|(m_i^e)^d\right|_n = \left|m_i^{de}\right|_n$$

Now we can use a connection of d, e and $\varphi(n)$:

$$d = \left|e^{-1}\right|_{\varphi(n)}$$
$$|d \cdot e|_{\varphi(n)} = 1$$
$$d \cdot e = k \cdot \varphi(n) + 1,\ \mathrm{k} \in \mathbb{N}$$

Getting back to our equation:

$$\left|m_i^{de}\right|_n = \left|m_i^{\varphi(n)\cdot k+1}\right|_n = \left|m_i^{\varphi(n)\cdot k} \cdot m_i^1\right|_n = \left|\left|m_i^{\varphi(n)\cdot k}\right|_n \cdot |m_i|_n\right|_n$$

If m_i and n are coprime, usage of *Theorem 2 (Euler's Theorem)* leads us to the wanted result.

$$\left|\left|m_i^{\varphi(n)\cdot k}\right|_n \cdot |m_i|_n\right|_n = \left|1^k \cdot m_i\right|_n = |m_i|_n = m_i$$

The last equation is a consequence of the condition $m_i < n$.

A problem arises in a situation when m_i and n are not coprime thus we would not be able to use Euler's theorem. There are $p-1$ multiples of q, $q-1$ multiples of p and 1, in total $p+q-1$ options for m_i not to be coprime with n. The chance of this happening is considered to be really low for huge prime numbers p and q, as shown below:

$$\lim_{(p,q)\to(\infty,\infty)} \frac{p+q-1}{n} = \lim_{(p,q)\to(\infty,\infty)} \frac{p+q-1}{p\cdot q} = \lim_{(p,q)\to(\infty,\infty)} \left(\frac{p}{p\cdot q} + \frac{q}{p\cdot q} - \right.$$
$$\left. - \frac{1}{p\cdot q}\right) = \lim_{(p,q)\to(\infty,\infty)} \left(\frac{1}{q} + \frac{1}{p} - \frac{1}{p\cdot q}\right) = \lim_{(p,q)\to(\infty,\infty)} \frac{1}{q} + \lim_{(p,q)\to(\infty,\infty)} \frac{1}{p} -$$
$$- \lim_{(p,q)\to(\infty,\infty)} \frac{1}{p\cdot q} = 0 + 0 - 0 = 0$$

Nonetheless, even if this situation occurs ($GCD\,(m_i, n) \neq 1$), RSA produces proper results, but proof of it is a bit complicated. Because only factors of n are p and q we can say for sure that $GCD\,(m_i, n) = q$ or $GCD\,(m_i, n) = p$. These cases are equivalent. Therefore, proof here will be done just for one of them, let's say $GCD\,(m_i, n) = p$.

$$m_i = l \cdot p,\; l \in \mathbb{N}$$

As a consequence of *Theorem 3 (Chinese Remainder Theorem)*:

$$\left|m_i^{de}\right|_p = |m_i|_p \wedge \left|m_i^{de}\right|_q = |m_i|_q \implies$$
$$\implies \left|m_i^{de}\right|_{pq} = |m_i|_{pq} = m_i$$

The first part of the condition is fulfilled easily because of $m_i = k \cdot p$. It gives us equation $0 = 0$, which is satisfied. The second part is proved by adjusting the expression m_i^{de}:

$$\left|m_i^{de}\right|_q = \left|m_i^{de-1} \cdot m_i\right|_q = \left|m_i^{\varphi(n)\cdot k} \cdot m_i\right|_q = \left|m_i^{(p-1)\cdot(q-1)\cdot k} \cdot m_i\right|_q =$$
$$= \left|\left(m_i^{(q-1)}\right)^{(p-1)\cdot k} \cdot m_i\right|_q = \left|1^{(p-1)\cdot k} \cdot m_i\right|_q = |m_i|_q$$

□

3.3 Security

It is widely believed that the security of RSA cryptosystem relies on the factorization of integers. But that has not been proven so far. It is unknown if an inverse of e modulo $\varphi(n)$ can be found without actually factoring number n. This is known as the *RSA problem*.

Chapter 4

Factoring algorithms

In this chapter, there are discussed some factorization techniques. In all the cases, let us assume that number given to factorize is not a prime, which can be determined with high precision (e.g., with Rabin-Miller's algorithm) in a short amount of time, nor simple squares of numbers. Checking these cases should be done before actually starting factorization but they are out of the scope of this thesis.

4.1 Trial division

The most well-known factorization algorithm, about whose even pupils in some elementary schools are taught, is the trial division.

The idea is straightforward. If we divide given number n by an integer i and the result is an integer, then i and n/i are some factors of n. Starting with $i = 2$, if the result is decimal, we raise i by one and try again. The upper bound of i is $\sqrt{n}$ from obvious reasons. If necessary, we can recursively apply this algorithm to already found factors if any of them is not a prime.

Clearly, there is some redundancy. If i is a composite number, there is no way it can divide our number n because we would already have tried to divide n by its factors. Thus only prime numbers are meaningful. However, determining if the number is prime will take some additional time. So there are two approaches.

The first one is the one mentioned earlier. We try to divide n by all numbers one by one. It is faster in total, but a bit inefficient. The second one will use a pre-generated list of prime numbers and try to divide n only by those. With that list already computed, this algorithm is faster. But when it is necessary to generate it first, it is in total slower.

As this algorithm is well known, there will be no example of its usage. Pseudocode *Algorithm 1* is here just for consistency.

Algorithm 1 Trial division

```
1: for $i \leftarrow 2$ to $\lfloor\sqrt{n}\rfloor$ do
2:     if $n/i$ is an integer then
3:         return $i$, $n/i$
4:     end if
5: end for
```

4.1.1 Complexity

This method can find low divisors of huge numbers very fast. But in the worst-case scenario, when n is a product of two prime numbers about the same size, $\mathcal{O}\left(\sqrt{n}\right)$ polynomial-time operations have to be executed. However, that is the dependence on the number n itself, not on the length of the input, which is in this case amount of bits of n, approximately $\log_2(n)$ bits. Therefore, to get input length n, the number taken into account should be 2^n, which makes complexity exponential, specifically $\sqrt{2^n}$ polynomial time operations.

The same logic of the exponential complexity from the dependence on the actual number n is used in some later chapters in a similar manner, as should be clear from the context.

4.2 Pollard's $p-1$ method

Pollard's $p-1$ method, invented by John M. Pollard in 1974, uses *Theorem 1 (Little Fermat's Theorem)*. Altering that formula leads to the key of this factorization principle. Let a be a small number greater than 1.

$$\begin{aligned} a^{(p-1)} &\equiv 1 \pmod{p} \\ a^{k\cdot(p-1)} &\equiv 1 \pmod{p},\ k \in \mathbb{N}^+ \\ a^{k\cdot(p-1)} - 1 &= l\cdot p,\ l \in \mathbb{N}^+ \end{aligned}$$

It can be applied in RSA schema in the following way. Modulus n can be decomposed as $n = p\cdot q$. If we use GCD, we can get to the following expression:

$$d = GCD\left(a^{k\cdot(p-1)} - 1,\ n\right) = GCD\left(l\cdot p,\ p\cdot q\right)$$

From what we can see now, $d = n$ in the case l is a multiplication of q or $d = p$. So if we construct expression $l\cdot p$ in a handy way, we get the prime factor p of the modulus n which is our primary task.

As a result, the problem has changed to find $b = a^{k\cdot(p-1)}$. Let's guess it. We evaluate $a^{f!}$ for $f = 2, 3, \ldots$ up to some limit. If that factorial contains $p-1$ value as well, we can obtain the value of p using GCD stated above. Let's examine possible results for $d' = GCD\left(a^{f!} - 1,\ n\right)$:

1. $d' = 1$

   This result indicates that our guess was unsuccessful. In other words, our product does not contain the required $p-1$ value. We can continue using this algorithm by increasing f and trying it again or just quit it.

2. $d' = n$

   In this situation, $f!$ contains both $p-1$ and $q-1$ (from *Theorem 2 (Euler's Theorem)*). Incrementing f is not much of a help because from now on the result will remain the same, n. In the situaion that f was not incremented just by one, a possible way is to reduce $f!$ by some part of the last increase and try to compute d' again.

3. d' is equal to some other value

   This case indicates success. It provides us notification that $f!$ in fact contains $p-1$. From the formula stated above, it is apparent that $d = p$, so we have our prime factor. The other one can be obtained by simple division $q = n/p$.

Algorithm 2 Pollard's $p-1$ algorithm using factorial

```
 1: choose upper bound $L$ and a small number $a$, $1 < a$
 2: $d \leftarrow GCD(a, n)$
 3: if $d > 1$ then
 4:     return $d$
 5: end if
 6: for $f \leftarrow 1$ to $L$ do
 7:     $b \leftarrow a^{f!}$
 8:     $d \leftarrow GCD(b-1, n)$
 9:     if $d = 1$ then
10:         continue
11:     else if $d = n$ then
12:         reduce $b$ by some number
13:         continue with step 8
14:     else
15:         return $d$
16:     end if
17: end for
```

Instead of using factorial, as it is in *Algorithm 2*, we can use another approach. The problem of using $f!$ is that soon we will get overcrowded by small prime numbers. There is a limit to each prime number how many times we really require it. Powers of primes, that exceed the value of n are obviously not useful. The maximal valuable exponent can be computed in the following way:

$$q_i^{e_i} \leq n$$
$$e_i \leq \log_{q_i} n = \frac{\ln n}{\ln q_i}$$

Algorithm 3 Pollard's $p - 1$ algorithm

```
1: choose the upper bound $L$ and a small number $b$, $1 < b$
2: $d \leftarrow GCD\,(b, n)$
3: if $d > 1$ then
4:     return $d$
5: end if
6: for each prime $q \leq L$ do
7:     $l \leftarrow q^{\lfloor \log_q n \rfloor}$
8:     $b \leftarrow b^l \pmod{n}$
9: end for
10: $d \leftarrow GCD\,(b - 1, n)$
11: if $d = 1$ then
12:     lift the bound $L$ and continue with step 6 or quit
13: else if $d = n$ then
14:     reduce $b$ by some number
15:     continue with step 10
16: else
17:     return $d$
18: end if
```

This approach can be found in *Algorithm 3*. There are two changes to be noticed. First, as discussed above, only the prime numbers and exponentiation are used. The second is that GCD is checked just once after the whole *for* loop is executed.

Pollard's original algorithm works for two cases. Let $1 < L < M < n^{\frac{1}{2}}$, $M < L^2$. It reveals all the factors of the form $p - 1 = A$ or $p - 1 = Aq$, where A is a multiplication of some primes not greater than L and q is one prime number $L < q < M$ [3, pp. 526–527]. The first case was discussed so far. The second one can be checked by the continuation of *Algorithm 3*. If a situation on line 11 happens, we can take value b and try to compute $b^m \pmod{n}$ for every prime $L < m < M$ and try to reveal factors of n by computing GCD again.

4.2.1 Complexity

Which of the versions above is faster depends on the actual number $p-1$. Factorial will have better performance for a multiplication of low primes to low powers, as the second one spends more time computing all the possible powers of low primes. For larger numbers, the situation changes because some low primes could be included more times than it is useful. In general, they are asymptotically very similar. Therefore, just the second version will be discussed.

For $p-1$ being *L-smooth* for some low L, we will get our result very quickly. On the other hand, if $p-1 = 2 \cdot p_p$ and $q-1 = 2 \cdot p_q$ where and p_p and p_q are primes about the same length, we have to iterate through the cycle over all primes up to the lower of $\{p_p, p_q\}$, which would give us the size of bound $L = \mathcal{O}\left(\sqrt{n}\right)$.

Complexity is given by the number of prime numbers up to the bound L. By the *Theorem 4 (Prime Number Theorem)*, this amount is about $\frac{L}{\ln L}$. Each iteration of the cycle can be done in polynomial time. This gets us to the final value

$$\mathcal{O}\left(\frac{L}{\ln L}\right) = \mathcal{O}\left(\frac{\sqrt{n}}{\ln \sqrt{n}}\right).$$

Thus, again exponential complexity, but slightly asymptotically faster than the trial division.

Example 1. *Let’s try to factorize $n = 18559$ using this algorithm. Let’s choose bound $L = 11$, $a = 2$. For all primes $p_i \leq 11$, compute corresponding maximal possible exponent e_i, $p_i^{e_i} < 18559$.*

$$2^{2^{14} \cdot 3^8 \cdot 5^6 \cdot 7^5 \cdot 11^4} \equiv 2480 \ (mod\ 18559)$$
$$GCD(2479, 18559) = 67$$

We have found one factor. The second can be gained by simple division $18559\ /\ 67\ =\ 277$.

4.3 Pollard’s rho method

Pollard’s rho algorithm is a currently used name for Pollard’s Monte Carlo factorization method firstly publicized by J. M. Pollard in 1975 in a mathematics journal BIT Numerical Mathematics in the article called “A Monte Carlo Method for Factorization”.

Let’s have a function $f(x)$ which will produce a pseudorandom sequence. With a finite set of numbers in modulus of n, after a maximum of n iterations of this function, we will get to a repetition of values, a cycle. There also can be some initial acyclic part called a tail. If we sketch it, it can give us a picture of Greek letter ρ, that is where the name of this algorithm came from.

The same sequence in modulus of p will cycle much faster since $p < n$. If we are capable of detecting a cycle in modulus of p, then we should be able to gain value of p easily. Original Pollard's algorithm uses *Floyd's cycle detection* [21, p. 176].

Theorem 6 (Floyd's cycle detection). *Let's have a sequence of numbers $(x_j)_{j\in\mathbb{N}^+}$ modulo p. If there is a cycle, then it will be certainly detected by the test $x_j \equiv x_{2j}$ (mod p) for each relevant j.*

Proof. Let us suppose that in our sequence there is actually a cycle with length l and acyclic part of length a, $l > 1$, $a \geq 0$. We use two pointers p and u that at the beginning both point to the first sequence element x_1. At each iteration, we move pointer p two elements forward (so in the first step to x_3) and u just to the next element, so indexes of the elements appointed by p will be two times bigger than the ones appointed by u. By some time (after $a - 1$ iterations) both pointers will point to elements in the cycle. Let y be a distance between them, specifically the distance from the pointer p to u in the direction of the cycle. We know that $y < l$. After each iteration from now on, y will be reduced by one. So after y iterations, both pointers will point to the same element, that is when $x_j \equiv x_{2j}$ occurs. □

But we are interested in the cycle modulo p without actually knowing p.

$$\begin{aligned} x_{2j} &\equiv x_j \pmod{p} \\ x_{2j} - x_j &\equiv 0 \pmod{p} \\ x_{2j} - x_j &= k \cdot p, k \in \mathbb{N} \end{aligned}$$

The beauty of this method is we actually do not need to know p to use a formula above. Let's compute $d = GCD\,(x_{2j} - x_j,\, n)$. There are several possible results for d:

1. $d = 1$

   This situation indicated that we were not able to get any useful results yet. It means the situation $x_j \equiv x_{2j}$ did not occur. Usually, we continue iterating $f(x)$.

2. $d = n$

   In this case, we return failure. No useful result is gained, we have to try variant seed values or/and different function in order to get demanded result [21, p. 181].

3. $1 < d < n$

   We have found two non-trivial factors of n, d and n/d.

The sequence in question is constructed by iterating function $f(x)$ for some seed value x over and over again $(f^j(x))$. But we do not need to store the whole sequence at once. We merely need to know the values of x_j and x_{2j}, next elements are simply computed as $x_{j+1} = f(x_j)$ and $x_{2\cdot(j+1)} = f(f(x_{2j}))$.

Now, let's have a look upon the function $f(x)$. It is widely spread that easy-to-compute function with sufficient randomness is the function $f(x) = x^2 + a$ for some $a \in \mathbb{N}^+ \setminus \{0, -2\}$. These two values cannot be used because of the stagnation of the algorithm. For $a = 0$, once we get to $x = 0$, we will not get any further. For $a = -2$ are the values in question $x = \pm 1$, as $(\pm 1)^2 - 2 = -1$ will cause stagnation as well. Original Pollard's algorithm uses $f(x) = x^2 - 1$ [4, p. 331]. Today is prevalent to use a function $f(x) = x^2 + 1$ as it is in *Algorithm 4*.

Computing *GCD* after each iteration of the function $f(x)$ is not that efficient and is not used in the original algorithm either. An often used improvement is to save intermediate results in a product and then compute *GCD* just once for the whole product, e.g., after 100 iterations [4, p. 331]. If we get $d = n$, we go back to the previous value and pick a lower step. Thus, it is substantial to save the last values of a (a_l) and b (b_l) for which we computed *GCD*. This approach can be found in *Algorithm 5*.

Algorithm 4 Pollard's rho algorithm without product

```
1: let's have $f(x) \leftarrow x^2 + 1$, $a \leftarrow 2$, $b \leftarrow 2$
2: $d \leftarrow 1$
3: for $i \leftarrow 1$ to $\infty$ do
4:     $a \leftarrow f(a) \pmod{n}$, $b \leftarrow f(f(b)) \pmod{n}$
5:     $d \leftarrow GCD\,(b - a,\, n)$
6:     if $1 < d < n$ then
7:         return $d,\ n/d$
8:     else if $d = n$ then
9:         return failure
10:    end if
11: end for
```

4.3.1 Improvement

Another improvement was brought by Richard P. Brent. He used a different cycle finding method which is about 36% faster than Floyd's and leads to approximately 24 % faster whole factorization algorithm [21, p. 176]. The purpose of this method is to reduce the amount of computing $f(x)$. It uses two pointer technique as well.

The principle is to move one pointer forward and then, after a decided amount of step (usually increasing powers of 2) we set the second to point to the corresponding element and start the cycle again. If it reaches the second

Algorithm 5 Pollard's rho algorithm

```
1: let's have $f(x) \leftarrow x^2 + 1$, $a \leftarrow 2$, $b \leftarrow 2$, $Q \leftarrow 1$, $s \leftarrow 100$
2: $a_l \leftarrow a$, $b_l \leftarrow b$
3: $d \leftarrow 1$
4: for $i \leftarrow 1$ to $\infty$ do
5:     $a \leftarrow f(a) \pmod n, b \leftarrow f(f(b)) \pmod n$
6:     $Q \leftarrow Q \cdot (b - a)$
7:     if $i$ mod $s = 0$ then
8:         $d \leftarrow GCD(Q, n)$
9:         if $1 < d < n$ then
10:            return $d$, $n/d$
11:        else if $d = n$ and $s \neq 1$ then
12:            $s \leftarrow 1$
13:            $a \leftarrow a_l$, $b \leftarrow b_l$
14:        else if $d = n$ then
15:            return failure
16:        end if
17:        $a_l \leftarrow a$, $b_l \leftarrow b$
18:    end if
19: end for
```

pointer before, then the cycle is there and we are done. As a bonus, we can easily gain loop length just by storing the number of steps done from the last meeting point. *Algorithm 6* below uses some common values. More information can be found in [21].

Algorithm 6 Brent's cycle-finding algorithm

```
1: $x \leftarrow 1$ $y \leftarrow x$, $r \leftarrow 1$, $k \leftarrow 0$, $j \leftarrow k$, done = false
2: while true do
3:     $r = r \cdot 2$
4:     $x \leftarrow y$, $j \leftarrow k$
5:     while $k \leq r$ do
6:         $k \leftarrow k + 1$
7:         $y = f(y)$, done $\leftarrow (x = y)$
8:         if done then
9:             return $l \leftarrow (k - j)$
10:        end if
11:    end while
12: end while
```

If we want to use Brent's cycle-finding method to speed up the factorization, we would need to alter it a bit similar way as Floyd's. It is even possible to use product improvement mentioned earlier.

4.3.2 Complexity

This algorithm has a weakness. Unlike the two previous methods, it is not certain it will discover any factor. The chance of failure lays in the $d = n$ situation discussed earlier. If all factors of n have the same cycle mod n, then this occurs all the time, and this method does not work. Another variable is a function $f(x)$. We do not know the value of p so we cannot be sure of its behaviour.

Let's check the complexity in the situation in which this method succeeds. That highly depends on the time to get a collision which we can get from the birthday paradox.

Theorem 7. *In Floyd's cycle-detection algorithm as discussed earlier, a collision usually appears after $\mathcal{O}(\sqrt{p})$ iterations of the function $f(x)$ assuming it behaves like a random function.*

Proof. There are p numbers in our set. We pick t of them, $1 \leq t \leq p$. To get the chance of the collision, we can apply the complement rule.

$$P_{collision} = 1 - P_{different}$$
$$P_{different} = \frac{p}{p} \cdot \frac{p-1}{p} \cdot \ldots \cdot \frac{p-t+1}{p} =$$
$$= 1 \cdot \left(1 - \frac{1}{p}\right) \cdot \left(1 - \frac{2}{p}\right) \cdot \ldots \cdot \left(1 - \frac{t-1}{p}\right)$$

The right side of the previous equation can be altered using definition of e^x.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$

Let's have x very small (as will turn out later). Then high order powers are so small we do not need to consider them.

$$e^x = 1 + x, \ x \ll 1$$

Using this equation, we can gradually get to our $P_{\mathit{different}}$.

$$\begin{aligned} e^{x} &= 1 + x \\ e^{-x} &= 1 - x \\ e^{-\frac{x}{p}} &= 1 - \frac{x}{p} \end{aligned}$$

$$\begin{aligned} P_{\mathit{different}} = 1 \cdot \left(1 - \frac{1}{p}\right) \cdot \left(1 - \frac{2}{p}\right) \cdot \ldots \cdot \left(1 - \frac{t-1}{p}\right) = \\ = e^{0} \cdot e^{-\frac{1}{p}} \cdot e^{-\frac{2}{p}} \cdot \ldots \cdot e^{-\frac{t-1}{p}} = e^{-\frac{0+1+2+\ldots+t-1}{p}} \end{aligned}$$

Now formula for a sum of the following numbers comes in hand.

$$\sum_{x=1}^{n} x = \frac{n \cdot (n+1)}{2}$$

$$P_{\mathit{different}} = e^{-\frac{0+1+2+\ldots+t-1}{p}} = e^{-\frac{\frac{(t-1)\cdot t}{2}}{p}} = e^{-\frac{t\cdot(t-1)}{2p}}$$

Let's see what happens if we want the probability of collision c, $0 < c < 1$.

$$\begin{aligned} P_{\mathit{different}} = e^{-\frac{t\cdot(t-1)}{2p}} &= 1 - P_{\mathit{collision}} = 1 - c \\ \frac{1}{e^{\frac{t\cdot(t-1)}{2p}}} &= 1 - c \\ e^{\frac{t\cdot(t-1)}{2p}} &= \frac{1}{1-c} \\ \frac{t\cdot(t-1)}{2p} &= \ln\left(\frac{1}{1-c}\right) \end{aligned}$$

For t large, $t - 1 \approx t$.

$$\begin{aligned} \frac{t^2}{2p} &= \ln\left(\frac{1}{1-c}\right) \\ t &= \sqrt{\ln\left(\frac{1}{1-c}\right) \cdot 2p} \\ t &= C \cdot \sqrt{p},\ C \in \mathbb{R}^{+} \end{aligned}$$

The last step is not entirely correct because the value of t does depend on the c, but as it is a square root of a logarithm, t grows very lightly with the rising value of c, therefore this approximation can be used. That means after $\mathcal{O}(\sqrt{p})$ picks from our set collision should occur. $\square$

Using the theorem above, a normal worst-case scenario has complexity $\mathcal{O}(\sqrt{p}) = \mathcal{O}\left(\sqrt[4]{n}\right)$ polynomial-time operations, which is significantly better than the trial division, but still exponential.

Example 2. *Let's try to factorize 18559. Used function will be $f(x) = x^2 + 1$ and start values $a = b = 2$.*

$1 : a = 2^2 + 1 = 5, b = (2^2 + 1)^2 + 1 = 26, Q = 26 - 5 = 21$

$2 : a = 5^2 + 1 = 26, b = (26^2 + 1)^2 + 1 = 12914, Q = 21 \cdot (12914 - 26) = 10822$

$\vdots$

$7 : a = 11915, b = 11512, Q = 9588$

$8 : a = 9435, b = 5884, Q = 9782$

$GCD\ (9782, 18559) = 67$

Is succeeded, factors of 18559 are 67 and 18559 / 67 = 277.

4.4 Fermat's factorization

Fermat's factorization method takes advantage of the following lemma.

Lemma 1. *Between any two various odd numbers p and q, let's say $p < q$, there always exists an odd integer x such as $x - p = q - x$. Its value is $x = \frac{p+q}{2}$.*

Let us suppose n we want to factorize is odd. Otherwise, we keep dividing it by two until it is. That means factors p and q we are looking for are odd as well. Thus, there exists a middle number x as mentioned in the *Lemma 1*. Let y be the distance of factors p and q from their average, $y = x - p = q - x$.

$$n = p \cdot q = (x - y) \cdot (x + y) = x^2 - y^2$$
$$y^2 = x^2 - n$$
$$y = \sqrt{x^2 - n}$$

Let us assume we somehow get the average x of the factors. Then we are able to get their distance and, as a result, both factors themselves. The point is how do we get the average. From the equation, it is obvious that $x \geq \sqrt{n}$. Fermat's algorithm then, as is shown in *Algorithm 7*, advances gradually through all numbers one by one until it finds the number x. That is notified by getting y as a whole number.

4.4.1 Improvement

In the 1920s Maurice Kraitchik came up with an enhancement that it is not necessary to look for x and y so $x^2 - y^2 = n$, but it is sufficient for it to be a multiple of n, that is, $x^2 \equiv y^2 \pmod{n}$ [2, p. 1474]. At least half of the

Algorithm 7 Fermat's factorization algorithm

```
1: $x \leftarrow \lceil \sqrt{n} \rceil$
2: while $\sqrt{x^2 - n}$ is not integer do
3:     $x = x + 1$
4: end while
5: return $x + \sqrt{x^2 - n},\ x - \sqrt{x^2 - n}$
```

solutions are interesting, which means $x \neq \pm y$. Then the prime factor of n can be found as $GCD\,(x - y,\ n)$. This is a very important expression that was at the basis of most modern factoring algorithms.

4.4.2 Complexity

The complexity of this algorithm highly depends on the distribution of the prime factors of n. If they are about the same size, this method reveals them almost instantly. However, in the worst case, when one of the factors is small, we get similar complexity as the trial division, specifically $\mathcal{O}\left(\sqrt{n}\right)$ polynomial-time operations.

For better usage, it can be combined with the trial division to decrease the possibility of low prime factors. This approach is usually faster than trial division or Fermat's factorization alone.

Example 3. *Let's try to factorize $n = 18559$ using this algorithm. Starting value is $x = \left\lceil \sqrt{18559} \right\rceil = 137$.*

$$\begin{aligned} x &= 137 & \sqrt{137^2 - 18559} &\approx 14.49 \\ x &= 138 & \sqrt{138^2 - 18559} &\approx 22.02 \\ &\vdots \\ x &= 172 & \sqrt{172^2 - 18559} &= 105 \end{aligned}$$

$$\begin{aligned} q &= x + \sqrt{172^2 - 18559} = 172 + 105 = 277 \\ p &= x - 105 = 67 \end{aligned}$$

Found factors are 67 *and* 277.

4.5 Dixon's random squares algorithm

In Fermat's factorization method, condition of $x^2 - n$ to be strait square is too strict. John D. Dixon has replaced it with the condition that $x^2 - n$ is ν-*smooth* for some chosen bound ν. By finding more numbers which fulfil this condition, using Gaussian elimination, we are able to construct a product of these in such a way that all powers of prime factors are even. Thus we can use a similar approach as in Fermat's algorithm to find factors of n.

Let us have set of prime numbers P with a length h containing all primes lower than a bound ν. For an integer x from interval $[1, n]$ [5, p. 256], if $x^2 \pmod{n}$ is *ν-smooth*, it can be written as a vector of exponents of primes p_i in the set P. For $\left|x^2\right|_n = p_1^{e1} \cdot p_2^{e_2} \cdots p_h^{e_h}$, vector of exponents would be $(e_1, e_2, e_3 \ldots, e_h)$. Those vectors are stored in the set B. Length of each of these vectors is h. Trial division is a sufficient tool for finding corresponding exponents, assuming not so large P. We are trying to find the first vector that is linearly dependent on the previous vectors. Using some knowledge of linear algebra, we need at most $h+1$ vectors. Dependency can be found using Gaussian elimination.

To be able to perform square root as in Fermat's approach, an exponent has to be even, so searching for linear dependency is sufficient to do modulo 2. Let c be the vector found dependent on some vectors $b \in B$. As it is not necessary for it to be dependent on all the previous, let f_b be the indicator (0 or 1) whether or not vector $b \in B$ is included.

$$c \equiv \sum_{b \in B} f_b \cdot b \pmod{2}$$

Therefore, $c + \sum_{b \in B} f_b \cdot b$ is an even number. Let

$$d = \frac{1}{2} \cdot \left(c + \sum_{b \in B} f_b \cdot b \right).$$

Just as c and b, variable d is a vector of exponents of primes from P. Let $y = \prod p_i^{d_i}$ for all primes p_i of P and their corresponding exponents (elements of d).

All numbers x_i whose powers of two were used to construct vectors from B (including c), whose $f_b = 1$, are stored in a set Z. Using $x = \prod_{x_i \in Z} x_i$ we get to congruence from the Fermat's algorithm, and we are done because

$$x^2 = \prod_{x_i \in Z} x_i^2 \equiv \prod p_i^{\left(c + \sum_{b \in B} f_b \cdot b\right)_i} = \prod p_i^{2d_i} = y^2 \pmod{n}$$

This algorithm can fail, that happens in the situation when $x = \pm y$. As pointed out in [5, p. 259], this chance is lower than $\frac{1}{2}$. Thus, the average amount of tries is lower than 2.

4.5.1 Complexity

Crucial is a choice of the bound ν but complete complexity analysis of Dixon's random square algorithm is beyond the scope of this thesis. Dixon in [5, p. 257] suggested $\nu = L(n)^{\sqrt{2}} = e^{\sqrt{2 \ln n \ln \ln n}}$ and proved the complexity of the algorithm to be $\mathcal{O}\left(L(n)^{3\sqrt{2}}\right)$. However, it has been proved in [22] that the best choice for ν is $L(n)^{\frac{1}{2}}$ with complexity $\mathcal{O}\left(L(n)^{2+\mathcal{O}(1)}\right)$.

4.6 Quadratic sieve

The quadratic sieve was made up in 1981 by Carl Pomerance. It uses a similar approach as Dixon's method, that is finding congruences of the shape $x^2 \equiv y^2 \pmod{n}$ and then uses technique mentioned earlier in Fermat's factorization.

The biggest difference from Dixon's algorithm is in finding numbers whose squares are smooth. Firstly, they are chosen relatively small compared to n, so there is a bigger chance of finding suitable values. Also, the sieving method is used to find them faster. The bound chosen is $\nu = L(n)^a$ for some a, $\frac{1}{10} < a < 1$ [6, pp. 131–132].

The main idea is the following. Let $S(A) = \left(\sqrt{n} + A\right)^2 - n$ for A being some small number, then the value of $S(A)$ is about $2\sqrt{n}A$. Moreover, a technique similar to the *Sieve of Eratosthenes* is used. Its validity is shown below. Let p be some prime number and k arbitrary integer.

$$\begin{aligned} S(A) &= \left(\sqrt{n} + A\right)^2 - n = n + 2\sqrt{n}A + A^2 - n = 2\sqrt{n}A + A^2 \\ S(A+kp) &= \left(\sqrt{n} + A + kp\right)^2 - n = n + 2\sqrt{n}(A+kp) + (A+kp)^2 - n = \\ &= 2\sqrt{n}A + 2\sqrt{n}kp + A^2 + 2Akp + (kp)^2 = S(A) + kp \cdot (2\sqrt{n} + \\ &+ 2A + kp) \end{aligned}$$

Thus, the following holds.

$$S(A) \equiv S(A+kp) \pmod{p}$$

Let us have a look at quadratic congruence $S(A) \equiv 0 \pmod{p}$. Let's mark unknown $x = \sqrt{n} + a$. Then it has at most two solutions.

$$\begin{aligned} S(A) = \left(\sqrt{n} + A\right)^2 - n &\equiv 0 \pmod{p} \\ x^2 &\equiv n \pmod{p} \end{aligned}$$

There is at least one solution if either $p|n$ or n is a quadratic residue modulo p. The situation $p|n$ is not interesting because it is very unlikely to happen and it can be checked very quickly. The second scenario suggests two solutions, $x \pmod{p}$ and $-x \pmod{p}$. However, that is true just for odd primes. If $p = 2$, these solutions are both equal to 1 as n is not divisible by p. As we want n to be quadratic residue modulo p, not all primes are useable for sieving. If a prime is useful is decidable quickly by *Euler's Criterion.* Let's start from *Theorem 1 (Little Fermat's Theorem).*

$$\begin{aligned} n^{p-1} &\equiv 1 \pmod{p} \\ \left(n^{\frac{p-1}{2}} + 1\right) \cdot \left(n^{\frac{p-1}{2}} - 1\right) &\equiv 0 \pmod{p} \end{aligned}$$

One of the factors must be equal to zero. If n is a quadratic residue, we can write $x^2 \equiv n \pmod{p}$.

$$\left(\left(x^2\right)^{\frac{p-1}{2}} + 1\right) \cdot \left(\left(x^2\right)^{\frac{p-1}{2}} - 1\right) \equiv 0 \pmod{p}$$
$$\left(x^{p-1} + 1\right) \cdot \left(x^{p-1} - 1\right) \equiv 0 \pmod{p}$$

From *Theorem 1 (Little Fermat's Theorem)* again, the second factor is zero. Thus, whether or not n is a quadratic residue modulo p can be discovered from the result of $n^{\frac{p-1}{2}} \pmod{p}$. This expression is called *Legendre symbol.* If it is equal to 1, then n is a quadratic residue.

Solving congruence $S(A) \equiv 0 \pmod{p}$ leads to seed value A_p (or more of them). For each suitable prime p, we get the whole bunch of values we can work with. If we have saved more values of $S(A)$, then by increasing value of k we get all $S(A)$ divisible by p. Smooth numbers can be found quickly this way. If we keep track for each $S(A)$ which primes p are its divisors, every $S(A)$ value which will be close to the multiplication of its found divisors would probably be $L(n)^a$*-smooth.*

There is one thing to be concerned about. Sieving process described above finds prime numbers which divide $S(A)$. However, it does not recognize whether or not $S(A)$ is divisible by the higher powers of these primes. In order to do that, two solutions can be applied. First is to check for the powers using trial division. The other one is to try sieving using some higher powers of primes, which is valid as well.

To speed up this algorithm even more, we can use negative values of x, which will allow us to have two times more numbers and thus lower $|S(A)|$, so operations will be performed faster. In order to do so, we need to include -1 in our list of suitable primes and work with it as with a regular prime in our set.

Another thing to decide is how many $S(A)$ values should be precomputed to get enough ν*-smooth* numbers. According to the Pomerance, if we have L^b values of $S(A)$, there are $L^{b-(4a)^{-1}}$ numbers which are composed solely of the primes $p < L^a$ [6, p. 132]. So the choice of $b = a + (4a)^{-1}$ seems reasonable to get enough values to find a linear dependency.

$$L^{b-(4a)^{-1}} = L^{a+(4a)^{-1}-(4a)^{-1}} = L^a$$

It is sumarized in *Algorithm 8.*

4.6.1 Improvement

There exists a variation of quadratic sieve called MPQS (multiple polynomial quadratic sieve), which uses more than one polynomial. They should be

Algorithm 8 Quadratic sieve

```
1: choose bound $L^a$, $\frac{1}{10} < a < 1$, $i \leftarrow 0$
2: create prime set $P$ of $p < L^a$ for which $n$ is quadratic residue modulo $p$
3: add $-1$ to the set $P$
4: for $A \in \{\pm1, \pm2, \pm3, \dots\}$ do
5:    compute $S(A) = (\sqrt{n} + A)^2 - n \pmod{n}$
6:    $i \leftarrow i + 1$
7:    if $i = L^{a+(4a)^{-1}}$ then
8:       break
9:    end if
10: end for
11: for each number $p \in P$ do
12:    find seed value/values $A_0$ such as $S(A_0) \equiv 0 \pmod{p}$
13:    starting at $A_0$, mark every $p$-th value of $S(A)$ as a multiplication of $p$
14:    if there are $|P| + 1$ (almost) completely factorized $S(A)$ values then
15:       break
16:    end if
17: end for
18: construct matrix of vectors of powers of primes mod 2
19: find linear dependency
20: construct expression $x^2 \equiv y^2 \pmod{n}$
21: if $x = \pm y$ then
22:    repeat algorithm with a different choice of bound
23: else
24:    return $GCD(x \pm y, n)$
25: end if
```

similar to the original $S(A)$. Its biggest advantage is in parallelization, one possible way is to provide each computer with different polynomial and sieve independently [2, p. 1479].

4.6.2 Complexity

As the theory for obtaining complexity of the quadratic sieve algorithm is beyond the scope of this thesis, let's just say that complexity is

$$\mathcal{O}(L(n)^{\frac{r}{\sqrt{4r-4}}+\mathbb{O}(1)}) = \mathcal{O}\left(e^{\left(\frac{r}{\sqrt{4r-4}}+\mathbb{O}(1)\right)\cdot\sqrt{\ln n \ln\ln n}}\right),$$

where r is an exponent of the complexity of used elimination algorithm. Often used Gaussian elimination has complexity t^3 for a square matrix with t rows. Thus, for $r = 3$ we get

$$\mathcal{O}\left(e^{\left(\frac{3}{\sqrt{4\cdot3-4}}+\mathbb{O}(1)\right)\cdot\sqrt{\ln n \ln\ln n}}\right) = \mathcal{O}\left(e^{\sqrt{\ln n \ln\ln n}}\right) = \mathcal{O}\left(L(n)\right).$$

Example 4. *Let's try to factorize* $n = 18559$ *using simplified version of this algorithm. Starting value is* $\left\lfloor \sqrt{18559} \right\rfloor = 136$. *Assume that the chosen prime set is* $P = \{2, 3, 5, 7, 11\}$. *The sieving process is omitted due to small numbers.*

$$\begin{gathered}
(136+1)^2 - 18559 = 210 = 2 \cdot 3 \cdot 5 \cdot 7 \\
(136-1)^2 - 18559 = -334 = -1 \cdot 2 \cdot 167 \\
(136+2)^2 - 18559 = 485 = 5 \cdot 97 \\
\vdots \\
(136+7)^2 - 18559 = 1890 = 2 \cdot 3^3 \cdot 5 \cdot 7
\end{gathered}$$

As the values are small, dependence can be seen immediately.

$$\begin{gathered}
137^2 \cdot 143^2 \equiv 2^2 \cdot 3^4 \cdot 5^2 \cdot 7^2 \ \textit{(mod n)} \\
(137 \cdot 143 - 2 \cdot 3^2 \cdot 5 \cdot 7) \cdot (137 \cdot 143 + 2 \cdot 3^2 \cdot 5 \cdot 7) \equiv 0 \ \textit{(mod n)} \\
402 \cdot 1662 \equiv 0 \ \textit{(mod n)}
\end{gathered}$$

$$\begin{gathered}
GCD(18559, 402) = 67 \\
GCD(18559, 1662) = 277
\end{gathered}$$

Found factors are 67 *and* 277.

4.7 Others

There are so many other factorization algorithms that were not discussed. A lot of them are just variations or modifications of the methods described. Some are significantly different. Here is a brief overview of some important algorithms which were not described earlier.

Hugh C. Williams in 1982 presented *"A* $p + 1$ *Method of Factoring"*. As the name suggests, it is an analogous method to Pollard's $p - 1$ factorization algorithm, working well if, for factor p of integer n, $p + 1$ is a *smooth* number.

Elliptic curve factorization method can be considered for generalization of Pollard's $p - 1$ algorithm in the terms that factorization is not done on the group $\mathbb{Z}_p$, but on group of points of random elliptic curve over $\mathbb{Z}_p$. If the order of this group is *smooth* for some chosen bound B, non-trivial factors of n can be gained.

The currently fastest general-purpose factoring algorithm is called *general number field sieve*. It is based on the idea of John Pollard from 1988 to use algebraic number fields. As it was originally meant for factoring numbers of special shape, it has developed to the general-purpose algorithm with expected running time $e^{c \cdot (\ln n)^{\frac{1}{3}} (\ln \ln n)^{\frac{2}{3}}}$. It believed that for numbers smaller than 100 digits, the *quadratic sieve* is faster. But for numbers of 130 and more digits, *number fields sieve* is better [2, p. 1483]. In fact, it is still the fastest applicable algorithm known.

The last method to be mentioned here is *Shor's algorithm.* It is the reason why in the era of quantum computing RSA does not stand a chance. Factoring can be done in polynomial time. This algorithm finds the least integer r such as $x^r \equiv 1 \pmod{n}$. Then computing $GCD(x^{\frac{rx}{2}} - 1, n)$ gives us a factor of n with a probability higher than $\frac{1}{2}$ assuming n has at least two different prime factors [23, p. 130].

It is really hard to guess the speed of development in the area of quantum computing. It is possible that some breakthrough will happen soon and quantum computers will become widely spread devices in the near future. But it is not considered to be that likely. According to the estimate mentioned in [24, 1:44:50], amount of qubits needed to factorize numbers used in RSA, which is more than 10^8 qubits [24, 1:15:42], compared to 53 qubits that are reached as of today, will be gained about the year 2050, so it is not something we should be concerned about now.

Chapter 5

Attacks on RSA

Exposing the private key from the public one is not an easy task. It is known as *the RSA problem*. It is widely believed that it is computational complexity is the same as of the integer factorization although it has not been proven (or disproven) yet. Till it is solved, factorization is the only general way how to obtain the private key.

However, inappropriate settings of RSA parameters may end up in creating some vulnerabilities which then could be exploited. Chosen parameters are public modulus n (primes p and q) and public exponent e.

In this chapter, there will not be discussed attacks using factorization including poor choice of n that could lead to a quick discovery of primes p and q and thus, breaking the whole encryption system.

5.1 Common modulus

Let us suppose that we choose only one public modulus for all users. To be able to use encrypted communication, each is given a unique pair of e_i and d_i using the same public modulus n. The reason of this could be, e.g., lower computational complexity compared to computing and storing various n_i.

This is an ultimately bad idea. Public modulus is as well as all public exponents accessible by anyone. Everybody who holds a pair, (e_i, d_i) can compute prime factors of n, as was shown in [25, p. 205], in the following way.

$$e \cdot d \equiv 1 \pmod{\varphi(n)}, \quad \varphi(n) = (p-1) \cdot (q-1)$$

then $k = ed - 1$ is a multiple of $\varphi(n)$. As $p - 1$ and $g - 1$ are both even, k is even as well. For $g > 0$, $GCD(g, n) = 1$ *Theorem 2 (Euler's Theorem)* applies:

$$g^k \equiv 1 \pmod{n}$$

Number $x = g^{\frac{k}{2}}$ is then square root of 1 modulo n. Let's rewrite it.

$$\begin{gathered}
x^2 \equiv 1 \pmod{n} \\
x^2 - 1 \equiv 0 \pmod{n} \\
(x+1) \cdot (x-1) \equiv 0 \pmod{n} \\
(x+1) \cdot (x-1) = l \cdot p \cdot q,\ l \in \mathbb{N} \\
\frac{(x+1) \cdot (x-1)}{p \cdot q} = l
\end{gathered}$$

This equation has 4 solutions. For $l = 0$, there are two trivial solutions, $x = \pm 1$. If $l > 0$, then

$$p|(x+1) \wedge q|(x-1) \quad \text{or} \quad p|(x-1) \wedge q|(x+1)$$

The other two solutions are

$$\begin{gathered}
x \equiv -1 \pmod{p} \quad \wedge \quad x \equiv 1 \pmod{q} \\
\mathit{and} \\
x \equiv 1 \pmod{p} \quad \wedge \quad x \equiv -1 \pmod{q}.
\end{gathered}$$

Computing $GCD(x-1, n)$ could reveal one of the factors of n. If it does not and $k/2$ is still even number, we can divide it again trying to get yet another square root of unity. This step can be done several times till $k/2^i$ is even. If g is chosen randomly, solutions would be random as well. Therefore, probability of correct choice of g can be expected about 1/2. If continued division technique is used, then probability can get higher than 50%.

Algorithm 9 Common modulus attack

```
1: choose random low $g$, $2 < g < n - 2$
2: if $GCD(g, n) \neq 1$ then
3:   return $GCD(g, n)$
4: end if
5: $k \leftarrow e \cdot d - 1$
6: while $k \equiv 0 \pmod{2}$ do
7:   $x \leftarrow g^{\frac{k}{2}} \pmod{n}$
8:   $res = GCD(x - 1, n)$
9:   if $1 < res < n$ then
10:     return res
11:   end if
12:   $k \leftarrow k/2$
13: end while
14: go back to step 1
```

5.2 Same message attack

This attack is strongly connected with the previous one. Let's assume that the message m is encrypted by two different public keys e_1 and e_2. That can happen in a common modulus scenario but also when some kind of error occurs during communication, so the public exponent is changed and the same message is transmitted again. If $GCD(e_1, e_2) = 1$, then plaintext can be obtained as was shown in [26, p. 181].

$$\begin{aligned} m^{e_1} &\equiv c_1 \pmod{n} \\ m^{e_2} &\equiv c_2 \pmod{n} \end{aligned}$$

There exists *Bézout's identity* of form $ae_1 + be_2 = 1$, where one of a, b is a negative and one is a positive integer. They can be found by the *Euclidean algorithm.* The inverse of ciphertext modulo n whose corresponding exponent has negative coefficient is computed, let's say $b < 0$, then $c_2^{-1} \pmod{n}$. All computations are done modulo n.

$$c_1^a \cdot \left(c_2^{-1}\right)^{|b|} = (m^{e_1})^a \cdot \left((m^{e_2})^{-1}\right)^{|b|} = m^{ae_1} \cdot m^{be_2} = m^{ae_1+be_2} = m^1 = m$$

Thus, anybody has access to the plaintext m. However, prevention against this attack is very simple. Sufficient is to change the message somehow, e.g., by adding timestamp or different padding.

Algorithm 10 Same message attack

```
1: find $a$ and $b$ such as $a \cdot e_1 + b \cdot e_2 = 1$
2: $m_1 \leftarrow c_1^a \pmod{n}$
3: $m_2 \leftarrow c_2^b \pmod{n}$
4: $message \leftarrow m_1 \cdot m_2 \pmod{n}$
5: return $message$
```

5.3 Low private exponent

If parameters of RSA are chosen in a way that the private exponent d is small, e.g., for speeding up decryption, there exists a cryptanalytic attack on the RSA, which can reveal factorization of the public modulus n. It was described by Michael J. Wiener in [27]. To get there, let's start from the relation between the public and private exponents.

$$\begin{gathered} ed \equiv 1 \pmod{\varphi(n)} \\ ed = k \cdot \varphi(n) + 1,\ k \in \mathbb{N} \\ ed = k \cdot (p-1) \cdot (q-1) + 1 \\ \frac{e}{pq} = k \cdot \frac{pq - p - q + 1}{pqd} + \frac{k}{kpqd} = \frac{k}{d} \cdot \left(1 - \frac{p + q - 1 - \frac{1}{k}}{pq}\right) \end{gathered}$$

Let's mark $\delta = \frac{p+q-1-\frac{1}{k}}{pq}$. It can be rewritten as

$$\frac{e}{pq} = \frac{k}{d} \cdot (1-\delta).$$

This equation can be solved for low δ with the *continued fraction algorithm* [27, p. 554], as $\frac{e}{pq}$ contains only public information and it is a close underestimate of $\frac{k}{d}$ for small δ. The algorithm can find values of k and d. Whether it succeeded or not can be checked quickly.

$$\begin{aligned} ed &= k \cdot (p-1) \cdot (q-1) + 1 \\ \frac{ed-1}{k} &= (p-1) \cdot (q-1) = \varphi(n) \end{aligned}$$

So we would know the value of $\varphi(n)$. Therefore, we are able to get both factors.

$$(p-1) \cdot (q-1) = pq - p - q + 1 \implies pq - (p-1) \cdot (q-1) + 1 = p + q$$

The same approach as in the *Fermat's factorization* can be used, in reverse order, to obtain values of p and q.

$$\begin{aligned} \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2 &= n \\ \left|\frac{p-q}{2}\right| &= \sqrt{\left(\frac{p+q}{2}\right)^2 - pq} \end{aligned}$$

If any of these steps returns an unexpected or an unwanted result, our guess for values d and k was not correct, and we have to repeat it in order to succeed.

Now about the actual *continued fractions algorithm* to get values d and k. A continued fraction is an expression of the form

$$\frac{a_1}{q_1 + \frac{a_2}{q_2 + \frac{a_3}{\vdots}}}.$$

If all a_i are equal to 1, we are interested only in the vector of q_i. Expansion of $\frac{e}{pq}$ will produce coefficients q_i. Along with that, we construct new fraction one step at time, the next coefficient is $q_i + 1$ if i is even or q_i if i is odd. From this fraction, we are able to reconstruct nominator and denominator of the original expression. After each iteration, check if $\frac{k}{d}$ is found.

It was shown that condition

$$dk < \frac{pq}{\frac{3}{2} \cdot (p+q-1)}$$

is sufficient to allow d to be found. With the typical case of RSA parameters, it will succeed for $d < n^{0.25}$ [27, pp. 557–558]. Nowadays, instead of *Euler's function*, *Carmichael function* is used as a modulus to compute the second key from the first. The biggest difference in the procedure is that multiplication $(p-1) \cdot (q-1)$ is replaced by the expression $LCM(p-1, q-1)$, which makes it a bit complicated. More about it can be found in [27].

Algorithm 11 Low private exponent

```
1: while $\frac{k}{d}$ is not found do
2:   generate next quotient $q_i$ of continued fraction expansion of $\frac{e}{n}$
3:   if $i \equiv 0 \pmod 2$ then
4:     construct fraction from quotients $q_0, q_1, \ldots, q_i + 1$
5:   else
6:     construct fraction form quotients $q_0, q_1, \ldots, q_i$
7:   end if
8:   if constructed fraction is equal to $\frac{k}{d}$ then
9:     return $\frac{k}{d}$
10:  end if
11: end while
```

5.4 Broadcast attack

The broadcast attack, presented by Johan Hastad in [28], is a way to recover a message encrypted using a small public exponent, which is sent to more subjects.

Let us assume message m, which we want to send to more subjects, each with different RSA key (modules n_1, n_2, $\ldots$). These modules are coprime. Otherwise, it is possible to factor some of them, and this attack does not have to be used. Let's suppose all subjects use the same low public exponent e. Then, we can recover m^e using *Theorem 3 (Chinese Remainder Theorem)*. The result will be gained modulo $n_1 \cdot n_2 \cdots$. If e is lower than the number of messages, then $m^e < n_1 \cdot n_2 \cdots$. Thus, modulo reduction does not apply and m can be recovered by eth root.

This attack can be generalized for usage with more than just one e but it is beyond the scope of this thesis.

Algorithm 12 Broadcast attack

```
1: create system of $e$ linear congruences $c_i \equiv m^e \pmod{n_i}$
2: solve this system using Chinese remainder theorem for variable $c$
3: return $\sqrt[e]{c}$
```

5.5 Simple power analysis

This attack is aimed more to the bad implementation than the wrong usage. If the decryption process is not implemented correctly, it can lead to revealing private modulus d by side channel, power analysis attack.

In RSA, plain text is gained from ciphertext in the following way.

$$m_i = \left|c_i^d\right|_n$$

The algorithm for quick exponentiation is called *Square and Multiply.* The first step is to convert d to binary number. With process from high order bits of d to lower, do the following. For the first 1, just enlist c_i to the result. For each following 0, square the result. For 1, square the result and multiply it with c_i. The following example ilustrates this method.

$$\begin{aligned} d = 13 = 1101,\ c = 5,\ n = 21,\ &\text{result will be in } x \\ \underline{1}101\colon &x = 5 \\ 1\underline{1}01\colon &x = x^2 \cdot 5 \quad x = |25 \cdot 5|_{21} = 20 \\ 11\underline{0}1\colon &x = x^2 \quad x = |20 \cdot 20|_{21} = 1 \\ 110\underline{1}\colon &x = x^2 \cdot 5 \quad x = 1^2 \cdot 5 = 5 \end{aligned}$$

If the algorithm for decryption is not implemented correctly, by power analysis, it can be determined which bit of d was 0 and which 1 as for multiplication is needed some additional computational effort. Thus, it would be possible to gain whole private exponent. Defence against this attack is quite simple, sufficient is to add some odd multiplication for zero-bits as well.

Chapter **6**

Testing

Mathematical expressions for the computational complexities of the factorization methods can provide a reasonable estimate of the running time. However, that stands only for the worst-case scenarios, the real running times can be different. In this section, selected methods are tested and compared in simulations performed by Magma software [29].

Two approaches in terms of the choice of the number to factorize are taken into account, as there are some recommendations regarding the choice of RSA public modulus n and its factors p and q presented by *NIST* in [18, pp. 50–53] and [19, pp. 33–35]. Comparison between running times of the factorization performed on the RSA modulus and random numbers with the corresponding length is also included.

6.1 Factorization functions

Magma offers several factorization functions, namely trial division (*TrialDivision*)[3], Pollard's rho (*PollardRho*), Pollard's $p-1$ (*pMinus1*), Williams's $p+1$ (*pPlus1*), Shanks's square forms (*SQUFOF*), Elliptic Curve Method (*ECM*), multiple polynomial quadratic sieve (*MPQS*), Number Field Sieve(*NFS*) and method for small Cunningham numbers (*Cunningham*). The last two are not used in the tests, mainly because they require parameters whose reasonable choice is beyond this thesis. Shanks's square forms can be understood as a practical implementation of Fermat's factorization method. There also exists a function called *Factorization*, which is a combination of the previously mentioned functions to achieve close to optimal performance [30, p. 302].

[3]name of the function in Magma, without parameters

	length of modulus in bits										
Function	20	30	40	50	60	70	80	90	100	110	120
Factorization	0	0.001	0	0.001	0.002	0.016	0.035	0.055	0.101	0.272	0.339
TrialDivision	0	0	0.01	0.26	7.97	1719.81					
PollardRho	0	0	0.002	0.008	0.021	0.126	0.592	2.677	18.829	126.579	362.689
pMinus1	0.258	0.001	5.181	0.018	0.037	9.307	0.624	15.834	15.108	16.255	17.002
pPlus1	1.348	0.003	0.002	0.03	0.127	0.275	1.102	1.885	28.818	29.472	80.683
SQUFOF	0	0	0	0.001	0.002	0.011	0.03	0.272	1.125	7.597	20.94
ECM	0.006	0.003	0.005	0.031	0.156	0.432	1.102	5.818	4.812	20.2	67.489
MPQS	0.008	0.035	0.026	0.025	0.021	0.033	0.043	0.065	0.092	0.124	0.162

Table 6.1: Factorization methods run times in seconds

	length in bits										
Function	20	40	60	80	100	150	200	250	300	350	400
Factorization	0	0	0.002	0.035	0.101	1.330	19.495	415.350	7174.535	183821.2	
MPQS	0.008	0.026	0.21	0.43	0.092	0.56	17.515	343.83	7289.405	122109.87	
FactorizationR	0	0	0.001	0.003	0.007	0.047	1.89	8.788	62.551	575.376	124.632
NFS					3.99		61.19	340.43	2247.44	18841.1	100253

Table 6.2: Factorization methods run times in seconds – higher modulus

6.2 Factorization

Comparison of the mentioned methods on the numbers satisfying conditions of creating RSA public modulus, except for the length, was performed for the modulus range between 20 and 120 bits. Its size is limited by Magma implementation, higher values are problematic for some functions. Results presented in Table 6.1 and Figure 6.1 were gained as the average of 10 runs of each function for each modulus length, with the exception of *TrialDivision*.

Comparison by obtained results between each other almost corresponds with the comparison by computational complexities. Nonetheless, it could not be compared numerically because theoretical running times are asymptotical and curve regressions provided strongly misleading values.

Notable are the running times of the Pollard's $p-1$ method, which highly depends on the properties of a particular integer that is factorized. The similar should work for Williams's $p+1$ method as well, but it is not so obvious from the results.

The second test was about numbers with higher bit length. RSA modules of size up to 400 bits were tested by three methods – Magma functions *Factorization* and *MPQS* mentioned earlier. The third was CADO-NFS [8] implementation of the Number Field Sieve(NFS). In addition, function *Factorization* was used to factorize a random number of the corresponding length. It is denoted as *FactorizationR*. Results are gathered in Table 6.2 and Figure 6.2.

Results show that restrictions on creating RSA modulus are really important as the factorization of a randomly chosen number of the same length

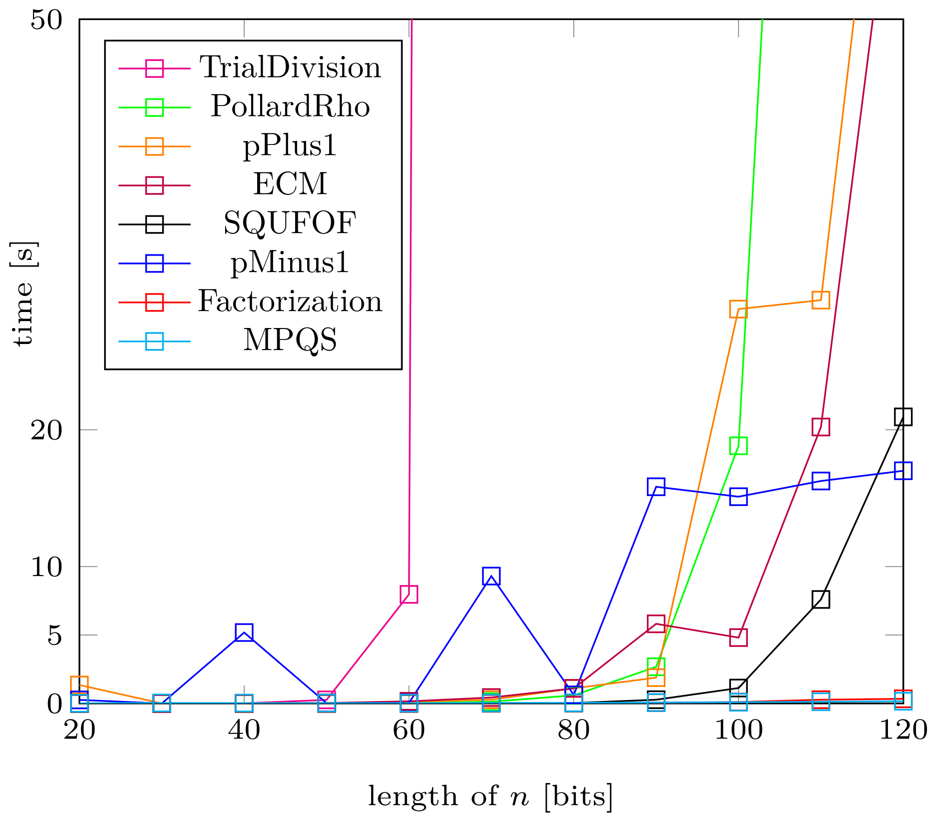

Figure 6.1: Factorization methods comparison – legend is sorted by run times

can be done much faster. *Factorization* and *MPQS* have very similar running times, which is not very surprising due to the fact that *Factorization* is mostly quadratic sieve [30, p. 302]. As the results confirm, NFS is, in fact, the fastest factorization algorithm for long RSA moduli.

6.3 Attacks

Results of testing of selected attacks on RSA excluding factorization are described in this section.

6.3.1 Low private exponent

Low private exponent attack was tested from the perspective of the relation between the maximal size of private exponent d and modulus n such as the factors of n can be revealed. Two attacks were performed – one using *Carmichael*

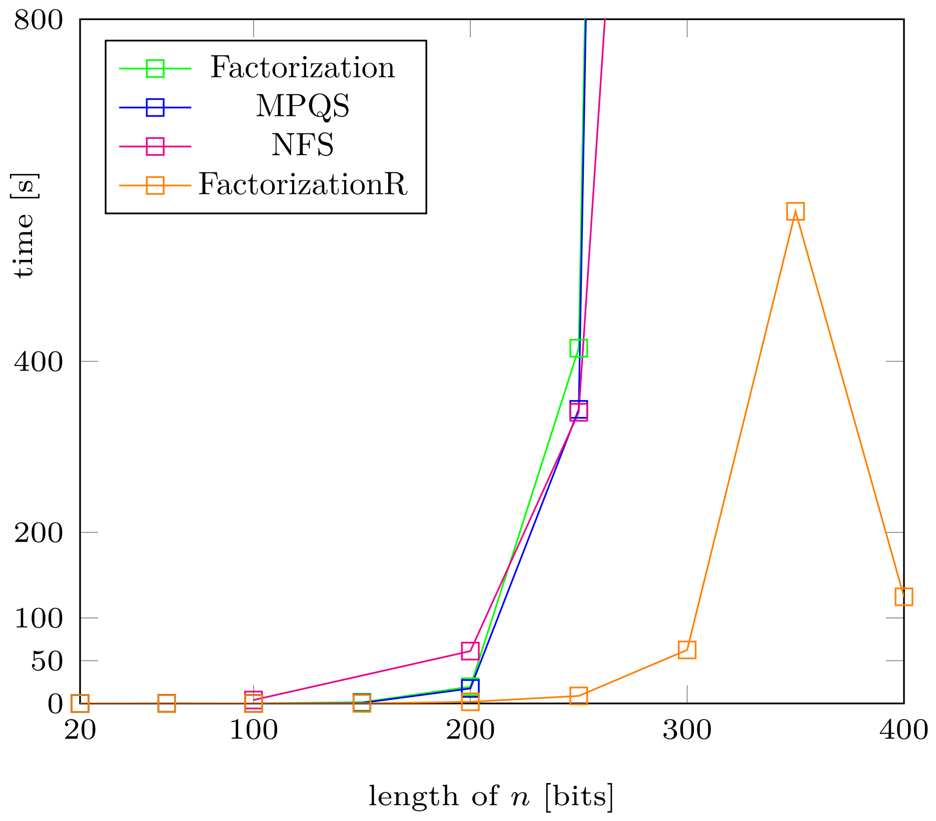

Figure 6.2: Factorization methods comparison – higher modulus

function (Figure 6.3) and the second using *Euler's function* (Figure 6.4) as the modulus for the computation of public exponent from the private one.

Let d_m be the maximal private exponent which is vulnerable to *low private exponent attack*. The results showed that $n^{0.246144} \leq d_m < n^{0.26209}$ for the *Euler's function* and $n^{0.240641} \leq d_m < n^{0.254649}$ for Carmichael function. Thus, for the prevention of this attack, private exponent should be chosen larger then $n^{0.25}$ unless another defence is chosen, as was suggested in [27, p. 557]

Edge values stated earlier were gained as the average of more than 200 runs of the attack for various bit lengths of the modulus and raising private exponent length by one at the time. How were these values distributed based on the modulus length is shown in Figure 6.3 and Figure 6.4.

6.3.2 Common modulus

Common modulus attack was performed on the modulus up to 2830 bits. Even for the numbers this large, the attack was able to recover private exponent d

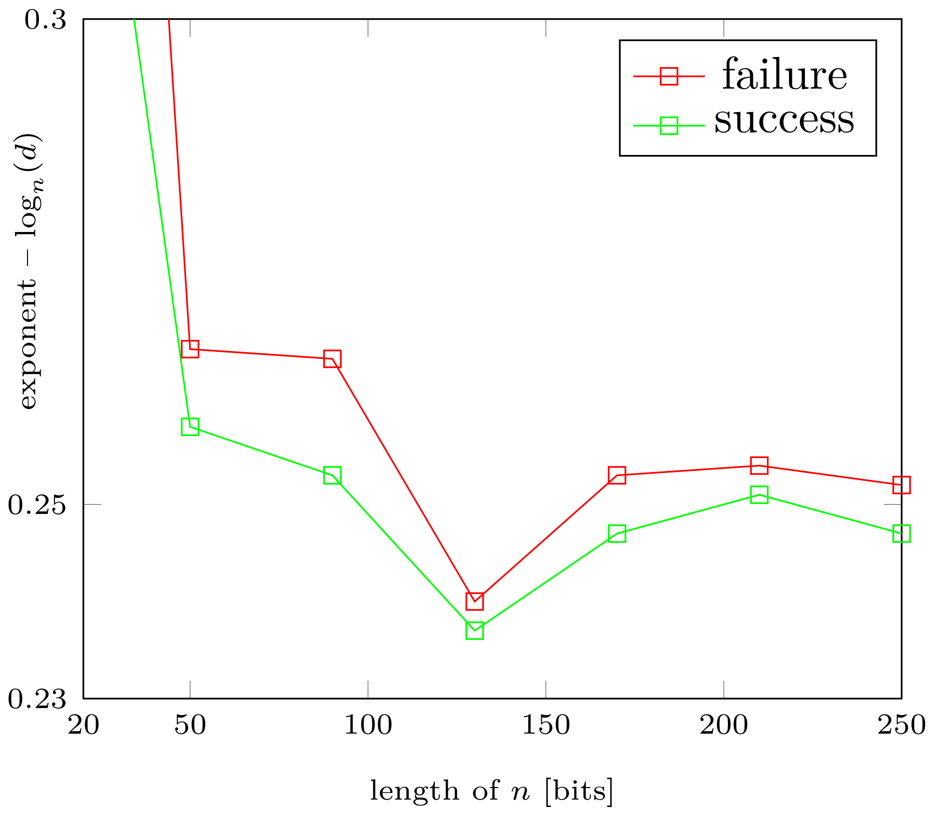

Figure 6.3: Low private exponent attack using Carmichael function

very quickly. In fact, for each size of the modulus, time needed to perform the attack was lower than 0.1 seconds. Thus, it is essential to be secure against this kind of attack. The test aimed at a number of choices of a random g to recover a private exponent. From 556 runs, 768 random choices of g were made, meaning 72.4 % success rate of the first choice. The average amount of division of k done for a choice of g was 2.457.

6.3.3 Same message

Same message attack is a speedy way to recover plaintext. All the attempts to perform this attack were from the time perspective at the edge of computer recognition (about 0.01 seconds). However, not all the attempts were successful. As was mentioned in description earlier, coprime public exponents are required. For a randomly chosen pair of public exponents on various modulus length, the attack was successful in 81.17% of cases from more than 1000 runs.

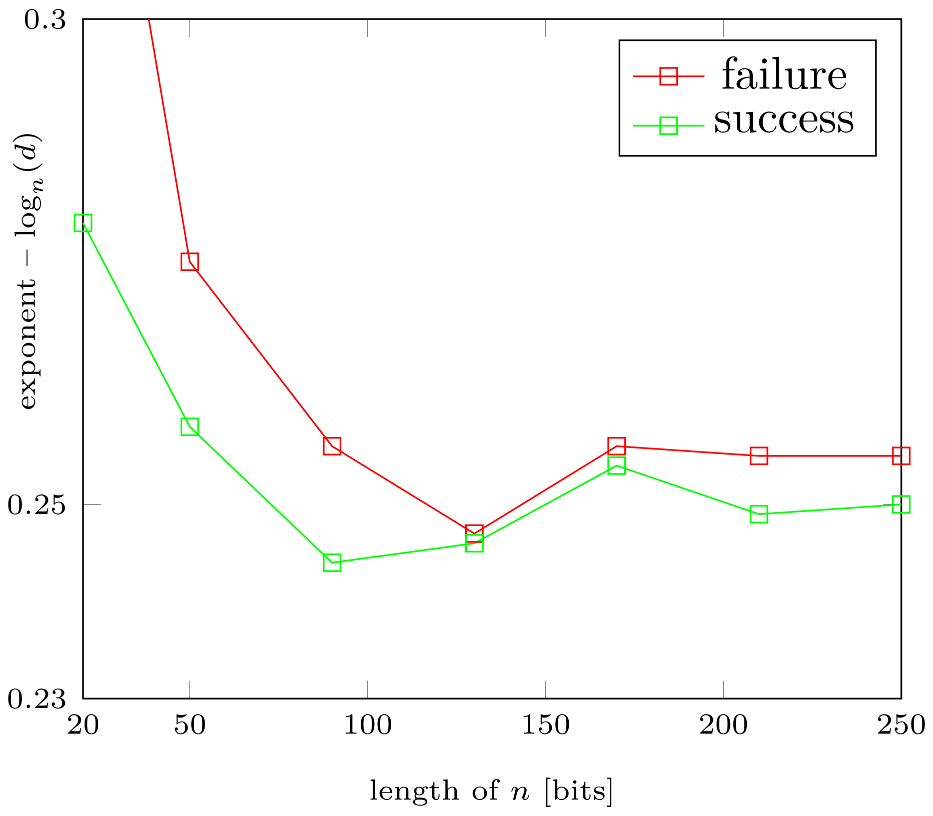

Figure 6.4: Low private exponent attack using Euler's function

6.3.4 Broadcast attack

As the size of the modulus and public exponent raises, the broadcast attack becomes slower and slower. Its computational complexity is exponential. Dependence of running time on the length of public exponent with fixed modulus length $n = 120$ bits is shown in Figure 6.5.

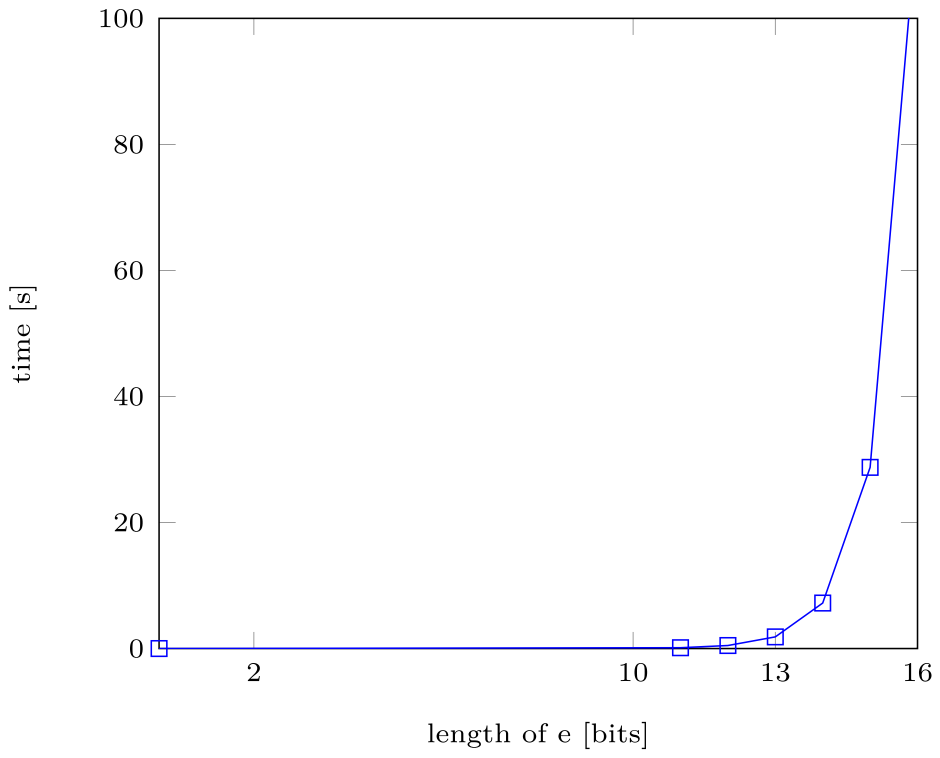

Figure 6.5: Broadcast attack for various public exponents

Conclusion

The goal of this thesis was to describe selected factorization methods and some attacks on RSA and test these. Provided descriptions and mathematical expressions were designed to help understand them in the prospective study, which I believe was achieved.

Tests of factorization algorithms were performed, and comparison of practical running times almost corresponds to the theoretical collation. Also, the impact on the whole cryptosystem in the case that modulus is chosen randomly and not according to recommendations was shown. Wrong usage or inappropriate parameters can lead to information disclosure or more severe problems using various attacks.

All the tests were performed in Magma. Even though there were some limitations concerning the accuracy of computations with huge numbers which led to some constraints during testing, Magma has proved to be a handy tool for mathematical computations.

Further works may follow on and focus on the algorithms omitted in this thesis and remaining attacks on RSA or their generalization. It is also possible to focus on creating new factorization methods. Nevertheless, as many great mathematicians already studied factorization problem, it is not very likely that computational complexity will get significantly better. However, a way of improvement can lay in the solution of the *RSA problem.* Maybe there exists a way to break this cryptosystem without factorization.

Bibliography

1. MOLLIN, Richard A. A Brief History of Factoring and Primality Testing B. C. (Before Computers). *Mathematics Magazine*. 2002, vol. 75, no. 1, pp. 18–29. Available from DOI: `10.1080/0025570X.2002.11953094`.
2. POMERANCE, Carl. A Tale of Two Sieves. *Notices of the AMS* [online]. 1996, vol. 43, pp. 1473–1485 [visited on 2020-04-07]. ISSN 0002-9920. Available from: `https://www.ams.org/notices/199612/pomerance.pdf`.
3. POLLARD, J. M. Theorems on Factorization and Primality Testing. *Mathematical Proceedings of the Cambridge Philosophical Society*. 1974, vol. 76, no. 3, pp. 521–528. Available from DOI: `10.1017/S0305004100049252`.
4. POLLARD, J. M. A Monte Carlo Method for Factorization. *BIT Numerical Mathematics* [online]. 1975, vol. 15, no. 3, pp. 331–334 [visited on 2020-04-07]. ISSN 1572-9125. Available from DOI: `10.1007/BF01933667`.
5. DIXON, John D. Asymptotically Fast Factorization of Integers. *Mathematics of Computation*. 1981, vol. 36, no. 153, pp. 255–260. Available from DOI: `10.1090/S0025-5718-1981-0595059-1`.
6. POMERANCE, Carl. Analysis and Comparison of Some Integer Factoring Algorithms. *Computational Methods in Number Theory*. 1982, no. 154, pp. 89–139.
7. LENSTRA JR., Hendrik W. Factoring Integers with Elliptic Curves. *The Annals of Mathematics*. 1987, vol. 126, no. 3, pp. 649–673. Available from DOI: `10.2307/1971363`.
8. THE CADO-NFS DEVELOPMENT TEAM. *CADO-NFS* [online] [visited on 2020-04-07]. Available from: `http://cado-nfs.gforge.inria.fr/`.

9. ZIMMERMANN, Paul. *Factorization of RSA-250* [online]. 2020 [visited on 2020-04-07]. Available from: `https://lists.gforge.inria.fr/pipermail/cado-nfs-discuss/2020-February/001166.html`.
10. BARKER, Elaine; ROGINSKY, Allen. Transitioning the Use of Cryptographic Algorithms and Key Lengths. *NIST Special Publication 800-131A Revision 2* [online]. 2019 [visited on 2020-04-07]. Available from DOI: `NIST.SP.800-131Ar2`.
11. MENEZES, Alfred J.; VAN OORSCHOT, P. C.; VANSTONE, Scott A. *Handbook of Applied Cryptography*. CRC, 1997. ISBN 0-8493-8523-7.
12. *Faktorizace velkých čísel* [online] [visited on 2019-12-21]. Available from: `http://artax.karlin.mff.cuni.cz/~ppri7485/nmib014/faktorizace.pdf`. [Translated by author].
13. RIESEL, Hans. *Prime Numbers and Computer Methods for Factorization* [online]. Birkhäuser Boston, 2012 [visited on 2020-04-07]. Modern Birkhäuser Classics. ISBN 978-0-8176-8298-9. Available from: `https://books.google.cz/books?id=94DaZuVETzIC`.
14. KOLÁŘ, Josef. *Properties of Primes* [online]. 2014 [visited on 2019-12-10]. Available from: `https://courses.fit.cvut.cz/BIE-ZDM/media/lectures/p11numbt2.pdf` [File available after login to CTU network – a copy of the file is on the enclosed CD].
15. KOLÁŘ, Josef. *Solving Linear Congruences* [online]. 2014 [visited on 2019-12-10]. Available from: `https://courses.fit.cvut.cz/BIE-ZDM/media/lectures/p12numbt3.pdf` [File available after login to CTU network – a copy of the file is on the enclosed CD].
16. CLAY MATHEMATICS INSTITUTE. *Millenium Problems* [online] [visited on 2020-04-07]. Available from: `https://www.claymath.org/millennium-problems`.
17. WEISSTEIN, Eric W. Prime Number Theorem. *MathWorld–A Wolfram Web Resource* [online]. 2003 [visited on 2020-04-07]. Available from: `https://mathworld.wolfram.com/PrimeNumberTheorem.html`.
18. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Digital Signature Standard (DSS). *Federal Information Processing Standards Publication (FIPS) 186–4*. 2013. Available also from: `https://doi.org/10.6028/NIST.FIPS.186-4`.
19. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. Digital Signature Standard (DSS). *Federal Information Processing Standards Publication (FIPS) 186–5 (Draft)*. 2019. Available also from: `https://doi.org/10.6028/NIST.FIPS.186-5-draft`.

20. RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM.* 1978, vol. 21, no. 2, pp. 120–126. ISSN 0001-0782. Available from DOI: `10.1145/359340.359342`.

21. BRENT, Richard P. An Improved Monte Carlo Factorization Algorithm. *BIT Numerical Mathematics.* 1980, vol. 20, no. 2, pp. 176–184. Available from DOI: `10.1007/BF01933190`.

22. KIMING, Ian. The ψ-function and the Complexity of Dixon's Factoring Algorithm [online]. 2005 [visited on 2020-04-08]. Available from: `http://web.math.ku.dk/~kiming/papers/2005_factoring/gamma.pdf`.

23. SHOR, Peter W. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science.* 1994, pp. 124–134. ISBN 0-8186-6580-7. Available from DOI: `10.1109/SFCS.1994.365700`.

24. MATEMATICKÉ PROBLÉMY NEMATEMATIKŮ. *Kvantový počítač – pátý jezdec apokalypsy? – Jiří Pavlů // Seminář MPN 20.11.2019* [online]. 2020 [visited on 2020-04-07]. Available from: `https://www.youtube.com/watch?v=7KNGYgiduoI&t=4473s` [Translated by author].

25. BONEH, Dan et al. Twenty Years of Attacks on the RSA Cryptosystem. *Notices of the AMS* [online]. 1999, vol. 46, no. 2, pp. 203–213 [visited on 2020-05-03]. ISSN 1088-6842. Available from: `https://www.ams.org/notices/199902/boneh.pdf`.

26. SIMMONS, Gustavus J. A "Weak" Privacy Protocol Using the RSA Crypto Algorithm. *Cryptologia.* 1983, vol. 7, no. 2, pp. 180–182. Available from DOI: `10.1080/0161-118391857900`.

27. WIENER, Michael J. Cryptanalysis of Short RSA Secret Exponents. *IEEE Transactions on Information Theory.* 1990, vol. 36, no. 3, pp. 553–558. Available from DOI: `10.1109/18.54902`.

28. HASTAD, Johan. Solving Simultaneous Modular Equations of Low Degree. *Siam Journal on Computing.* 1988, vol. 17, no. 2, pp. 336–341.

29. BOSMA, Wieb; CANNON, John; PLAYOUST, Catherine. The Magma algebra system. I. The user language. *J. Symbolic Comput.* 1997, vol. 24, no. 3-4, pp. 235–265. ISSN 0747-7171. Available from DOI: `10.1006/jsco.1996.0125`. Computational algebra and number theory (London, 1993).

30. CANNON, John; BOSMA, Wieb; FIEKER, Claus; STEEL, Allan. Handbook of Magma Functions [online]. 2013 [visited on 2020-05-05]. Available from: `https://www.math.uzh.ch/sepp/magma-2.19.8-cr/Handbook.pdf`.

Appendix **A**

Acronyms

FIT	Faculty of Information Technology
CTU	Czech Technical University in Prague
GCD	Greatest common divisor
LCM	Least common multiple
MPQS	Multiple polynomial quadratic sieve
NFS	Number Field Sieve
SQUFOF	Shanks's square forms factorization
ECM	Elliptic curve method
NIST	National Institute of Standards and Technologies

Appendix **B**

Contents of enclosed CD

```
readme.txt ....................... file with brief CD contents description
materials/ ....................... materials available from CTU network
thesis/ .......................................... text of the thesis
    thesis.pdf ................................ thesis in PDF format
    latex/ ........................................ LaTeX source files
src/ ................................... the directory of source codes
```